Session - 3 : REST Assured

1. Learning Objectives

By the end of this lesson, you will be able to:

- Understand the purpose and benefits of REST Assured for API testing.
- Set up REST Assured in a Java project.
- Use REST Assured to send HTTP requests and validate responses.
- Write test cases for REST APIs.
- Implement assertions to verify API response data.

2. Introduction

REST Assured is a Java library that simplifies testing RESTful APIs. It enables developers to validate HTTP responses and perform requests with minimal setup and code. In the context of test automation, REST Assured provides an intuitive syntax that abstracts much of the complexity associated with HTTP communication. Whether testing endpoints in isolation or as part of integration tests, REST Assured offers a streamlined approach for API testing in Java, enhancing reliability and speeding up test execution.

. Key Concepts and Definitions

- **REST (Representational State Transfer)**: An architectural style used for building scalable web services. RESTful APIs use HTTP methods like GET, POST, PUT, and DELETE to perform actions on resources.
- **API (Application Programming Interface)**: A set of rules and endpoints allowing software applications to communicate with each other.
- **HTTP Request**: A message sent by a client to a server to perform an action. In RESTful services, this action corresponds to CRUD operations.
- **Assertion**: A check that ensures the API response meets the expected criteria, such as specific status codes or response data.

Importance of REST Assured

REST Assured is a powerful tool for testing RESTful APIs. Its importance lies in:

- **Simplifying API Testing**: It abstracts complex HTTP communication, allowing developers to focus on writing meaningful tests rather than handling the intricacies of HTTP requests and responses.
- **Integration with Java**: Being a Java library, it integrates seamlessly with Java-based testing frameworks like JUnit and TestNG.
- **Support for Continuous Integration**: REST Assured can be easily integrated into CI/CD pipelines, enabling automated testing for APIs during development.

Advantages of REST Assured

1. **Intuitive Syntax**: The fluent interface makes it easy to read and write tests, enhancing productivity.
2. **Rich Assertion Support**: Built-in assertions and the ability to use Hamcrest matchers provide extensive validation options for responses.
3. **Support for Various Content Types**: Easily handles JSON, XML, and form data, making it versatile for different API responses.
4. **Detailed Reporting**: Supports logging requests and responses, which aids in debugging and understanding failures.

5. **Easy Setup**: Simple dependency management with Maven or Gradle allows quick setup.

Disadvantages of REST Assured

1. **Java Dependency**: As a Java library, it may not be suitable for projects using other programming languages.
2. **Learning Curve**: While the syntax is intuitive, new users may need time to understand all features and best practices.
3. **Limited to REST APIs**: It is primarily designed for RESTful services and may not support other architectures (e.g., SOAP) as effectively.

Features of REST Assured

- **HTTP Method Support**: Easily supports all HTTP methods (GET, POST, PUT, DELETE).
- **Response Validation**: Validate response codes, headers, and body content using assertions.
- **Path and Query Parameter Handling**: Simplifies the inclusion of dynamic parameters in requests.
- **Support for Authentication**: Handles various authentication mechanisms, including Basic Auth and OAuth.
- **Response Body Parsing**: Provides utilities for parsing and validating JSON and XML responses.

Limitations of REST Assured

- **Performance**: For very large-scale testing, it might not be as performant as some lightweight alternatives.
- **Limited to REST API Features**: Advanced features available in dedicated API testing tools (e.g., mocking, detailed performance testing) might be missing.
- **Dependency on Java Environment**: Requires a Java development environment, which may not be feasible for all projects.

5. Step-by-Step Explanation

Step 1: *Setting Up REST Assured in a Maven Project*

- To use REST Assured, first add it as a dependency in your project.
- If using Maven, add the following dependency to your `pom.xml` file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSch
ema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.or
g/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>geethu</groupId>
  <artifactId>geethu</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <dependencies>
      <!-- https://mvnrepository.com/artifact/io.rest-assured/rest-assured -->
<dependency>
    <groupId>io.rest-assured</groupId>
    <artifactId>rest-assured</artifactId>
    <version>4.5.1</version>
    <scope>test</scope>
</dependency>
<!-- https://mvnrepository.com/artifact/org.testng/testng -->
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.7.1</version>
    <scope>test</scope>
</dependency>
```

```
<!-- https://mvnrepository.com/artifact/com.googlecode.json-simple/json-simple -->
<dependency>
    <groupId>com.googlecode.json-simple</groupId>
    <artifactId>json-simple</artifactId>
    <version>1.1.1</version>
</dependency>
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.14.3</version>
</dependency>
<dependency>
   <groupId>io.rest-assured</groupId>
   <artifactId>json-path</artifactId>
   <version>4.5.1</version>
   </dependency>
<dependency>
   <groupId>io.rest-assured</groupId>
   <artifactId>xml-path</artifactId>
   <version>4.5.1</version>
   </dependency>
<dependency>
   <groupId>io.rest-assured</groupId>
   <artifactId>json-schema-validator</artifactId>
   <version>4.5.1</version>
   </dependency>
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.10.2</version>
    <scope>test</scope>
</dependency>


  </dependencies>
</project>
```

https://mvnrepository.com/artifact/org.testng/testng

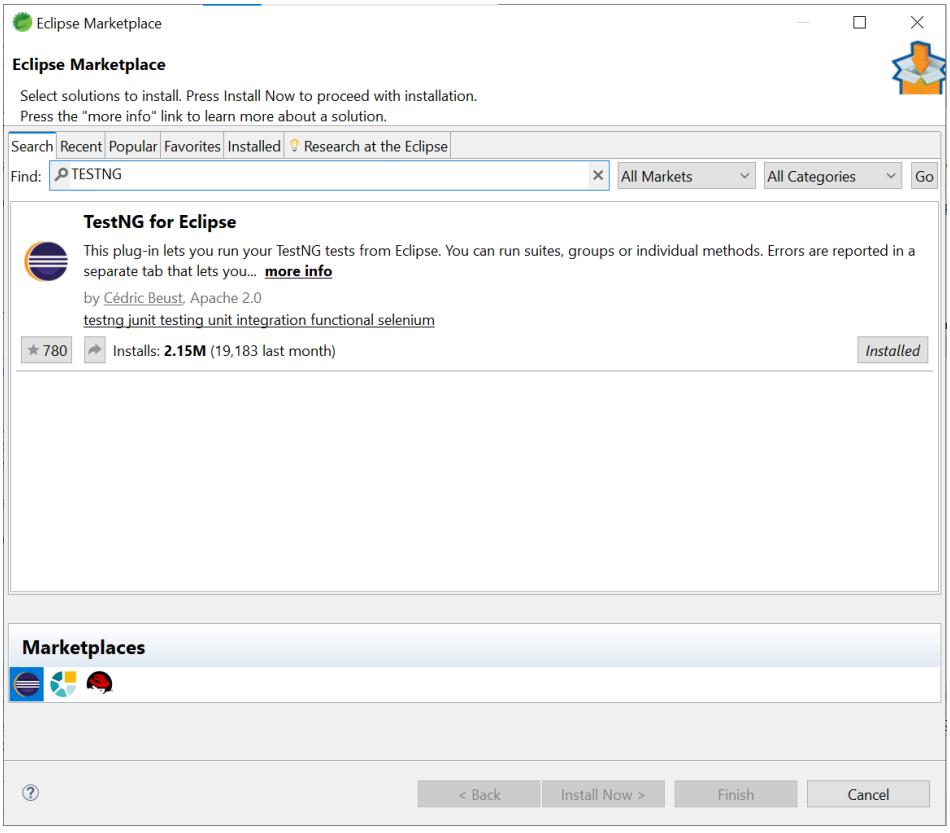
1. **Save** `pom.xml`:
   - Make sure any changes to your `pom.xml` file are saved. You can do this by clicking **File** > **Save** or using the shortcut (`Ctrl + S`).
2. **Right-Click on the Project**:
   - In the **Project Explorer** view, right-click on your project name.
3. **Go to Maven**:
   - In the context menu, hover over **Maven**.
4. **Update Project**:

- Click on **Update Project...** from the submenu.

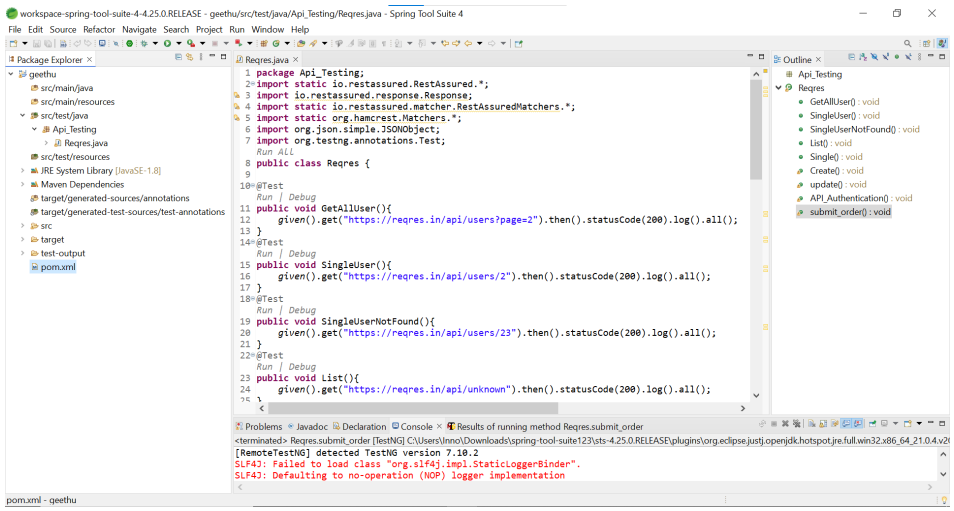
5. **Force Update of Snapshots**:
   - In the dialog that appears, check the box for **Force Update of Snapshots/Releases**.
6. **Click OK**:
   - Click the **OK** button to update the project. Eclipse will refresh the Maven dependencies based on your `pom.xml`.

Step 2: Install TestNG

1. **Open Eclipse Marketplace**:
   - Go to the **Help** menu in the top toolbar and select **Eclipse Marketplace...**.
2. **Search for TestNG**:
   - In the **Eclipse Marketplace** window, type **TestNG** in the search bar and press **Enter**.
3. **Click on Install**:
   - Find **TestNG for Eclipse** in the search results and click on the **Install** button next to it.
4. **Follow the Installation Wizard**:
   - Follow the prompts in the installation wizard. You may need to accept the license agreements and select any additional components.
5. **Restart Eclipse**:
   - Once the installation is complete, you may be prompted to restart Eclipse. Click **Yes** to restart.



File Structure of project



Step 2: *Basic REST Assured Request Structure*

- A typical REST Assured request follows the format:

```
given()
    .get("/endpoint")
    .then()
    .statusCode(200);
```

- `given()` specifies the request configuration, including headers, parameters, and authentication.
- `then()` is where you specify assertions, like verifying the HTTP status or checking response data.

Step 3: *Sending a GET Request*

- To fetch data from an API, use a GET request:

```
package Api_Testing;
import static io.restassured.RestAssured.*;
import io.restassured.response.Response;
import static io.restassured.matcher.RestAssuredMatchers.*;
import static org.hamcrest.Matchers.*;
import org.json.simple.JSONObject;
import org.testng.annotations.Test;
public class Reqres {

@Test
public void GetAllUser(){
    given().get("https://reqres.in/api/users?page=2").then().statusCode(200).log().all();
}
```

1. **Method Declaration**:
    - `public void GetAllUser()`: This defines a public method named `GetAllUser` that doesn't return any value (`void`).

2. `given()`:
    - This method is the starting point for building the request. It prepares the context for your request but here it's being used directly with a `get` call.
3. `.get("https://reqres.in/api/users?page=2")`:
    - This line makes a GET request to the specified URL. In this case, it's accessing the endpoint that returns a list of users from the `reqres.in` API. The parameter `page=2` indicates that you're requesting the second page of user results.
4. `.then()`:
    - This signifies that you're now defining what you expect from the response.
5. `.statusCode(200)`:
    - This assertion checks that the response has a status code of 200. A status code of 200 means that the request was successful, and the server has returned the expected data.
6. `.log().all()`:
    - This command logs the complete response, including headers, body, and other details, to the console. This is useful for debugging or verifying the data returned from the API.

Real-Life Example

Let's consider you are working on a project that requires retrieving user data for a web application. You need to ensure that the API you depend on returns the correct data. By running the `GetAllUser()` method:

- You will send a GET request to the endpoint `https://reqres.in/api/users?page=2`.
- If the API is functioning correctly and the request succeeds, you will receive a 200 OK status.
- The method will then log the full response to the console, allowing you to see all the details of the users returned in that response.

Imagine you're baking a cake. The recipe steps are like this:

1. **given()** – You gather your ingredients (flour, eggs, sugar).
2. **.get("/endpoint")** – You put your ingredients into the mixing bowl and start mixing.
3. **.then()** – You wait for the cake to bake in the oven.
4. **.statusCode(200)** – After the baking time is over, you check if the cake looks perfect, tastes good, and is ready to be served (successfully baked).

Step 4: *Sending a POST Request*

- For adding data, a POST request is used with REST Assured:

```
@Test
public void Create(){
    JSONObject js = new JSONObject();
    js.put("name", "morpheus");
    js.put("job", "leader");
    given().contentType("application/json").body(js.toJSONString()).when().post("https://re
qres.in/api/users").then().statusCode(201).log().all();
}
```

- Here, the request specifies a JSON body, and the response is validated to confirm the correct data was saved.
- `new JSONObject()` creates an empty JSON object.

- `JSONObject` is part of the `org.json` package (which you need to add as a dependency in your project if it's not already included). It allows you to work with JSON data in Java.
- The `put()` method is used to add key-value pairs to the JSON object. The first argument is the **key** (a String), and the second argument is the **value** (can be a String, number, boolean, or other data types).
- In this case:
  - `"name"` is the key, and `"morpheus"` is the value.
  - `"job"` is the key, and `"leader"` is the value.

Step 5: *Assertions in REST Assured*

- REST Assured offers built-in methods for assertions:
  - **statusCode()**: Checks the HTTP status code.
  - **body()**: Verifies specific fields in the response.
- Additional matchers like `equalTo()` and `notNullValue()` can be used from the `Hamcrest` library, which is often included with REST Assured.

6. Examples

- **Validating JSON Data**:

```
given()
    .baseUri("https://jsonplaceholder.typicode.com")
    .when()
    .get("/users/1")
    .then()
    .statusCode(200)
    .body("username", equalTo("Bret"))
    .body("address.city", equalTo("Gwenborough"));
```

- **Using Path Parameters and Query Parameters**:

```
given()
    .baseUri("https://api.example.com")
    .pathParam("id", 1)
    .queryParam("sort", "desc")
    .when()
    .get("/users/{id}/posts")
    .then()
    .statusCode(200);
```

Handling BEARER Authorization

Approach for OAuth Authorization in REST Assured:

1. **Obtain Access Token**: Send a request to the token endpoint to get an access token.
2. **Include Token in Requests**: Use the token in the Authorization header for subsequent API requests.

Example: Obtaining and Using an bearer Token

```
@Test
public void submit_order() {
```

```
    JSONObject js = new JSONObject();
    js.put("bookId", 1);
    js.put("customerName", "John");

    given()
        .header("Authorization", "Bearer " + "3a5d05c290a8244a9732200b39281db48dc2bba57a7b1da6
1ce7a0f798482eff")
        .contentType("application/json")
        .body(js.toString())
    .when()
        .post("https://simple-books-api.glitch.me/orders")
    .then()
        .statusCode(201)
        .log().all();
}
```

https://dummyjson.com/docs/auth

```
@Test
public void auth() {
    JSONObject js = new JSONObject();
    js.put("username", "emilys");
    js.put("password", "emilyspass");
    js.put("expiresInMins", 30);
    given()
        .contentType("application/json")
        .body(js.toString())
    .when()
        .post("https://dummyjson.com/auth/login")
    .then()
        .statusCode(200)
        .log().all();
}
```

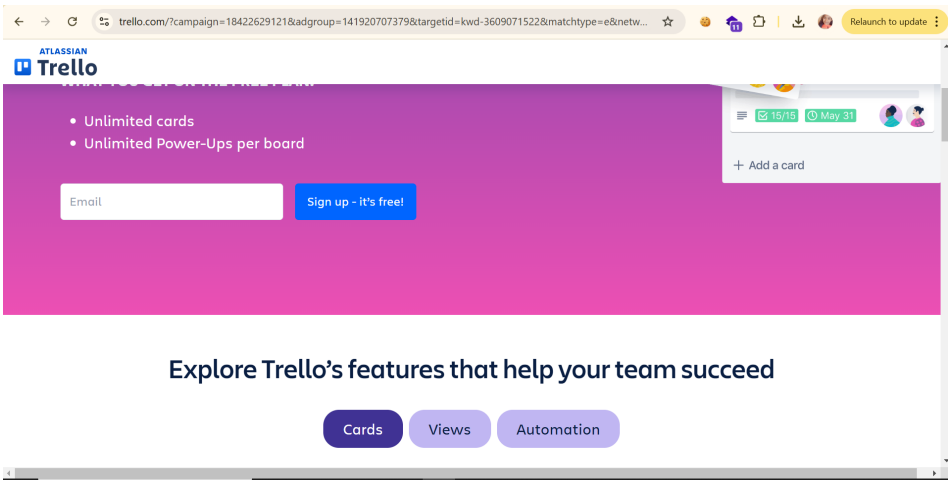
Let's relate this to a restaurant scenario where you're placing an order and making sure the restaurant knows exactly what you're asking for:

1. `given()` – Preparing to place the order.
    - You walk up to the counter (setting up your request) and are ready to tell the waiter what you want.
2. `.contentType("application/json")` – Telling the waiter the type of food you're ordering.
    - You tell the waiter, "I want an **Italian dish**," meaning you're ordering a dish that follows the Italian culinary tradition (just like specifying you're sending **JSON data**).
3. `.body(js.toString())` – Telling the waiter **exactly what you want to order**.
    - You hand the waiter a **recipe card** (your `js.toString()`), which lists the exact ingredients and preparation for the dish. The waiter reads the recipe card (the JSON string) to understand exactly what dish you want.

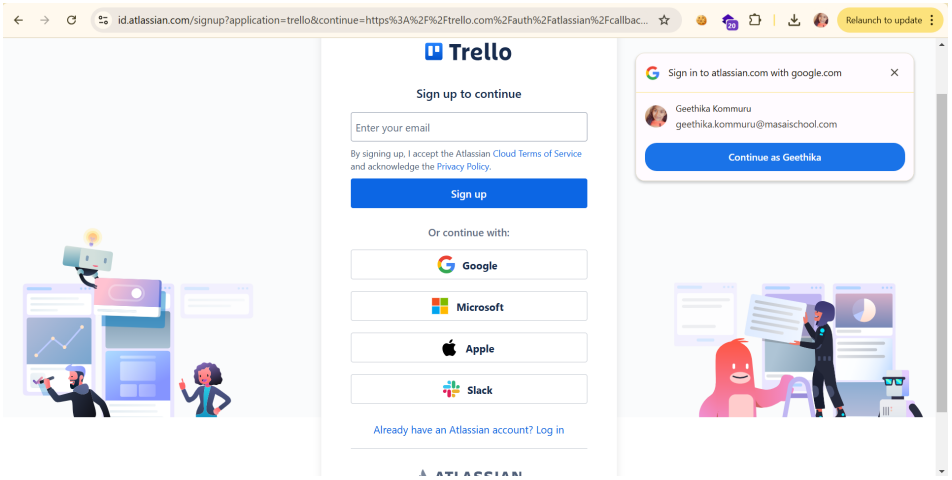
```
@Test
public void currentauth(){
    given()
    .header("Authorization", "Bearer " + "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwidXN
lcm5hbWUiOiJlbWlseXMiLCJlbWFpbCI6ImVtaWx5LmpvaG5zb25AeC5kdW1teWpzb24uY29tIiwiZmlyc3ROYW1lIjoiR
W1pbHkiLCJsYXN0TmFtZSI6IkpvaG5zb24iLCJnZW5kZXIiOiJmZW1hbGUiLCJpbWFnZSI6Imh0dHBzOi8vZHVtbXlqc29
uLmNvbS9pY29uL2VtaWx5cy8xMjgiLCJpYXQiOjE3MzA2MjY5NTgsImV4cCI6MTczMDYyODc1OH0.4nCfLW7v28wg_CLlt
gHgMPgK9T0GUk0SPT6eJukIg6I")
    .get("https://dummyjson.com/auth/me").then().statusCode(200).log().all();
}
```

How to Use Trello

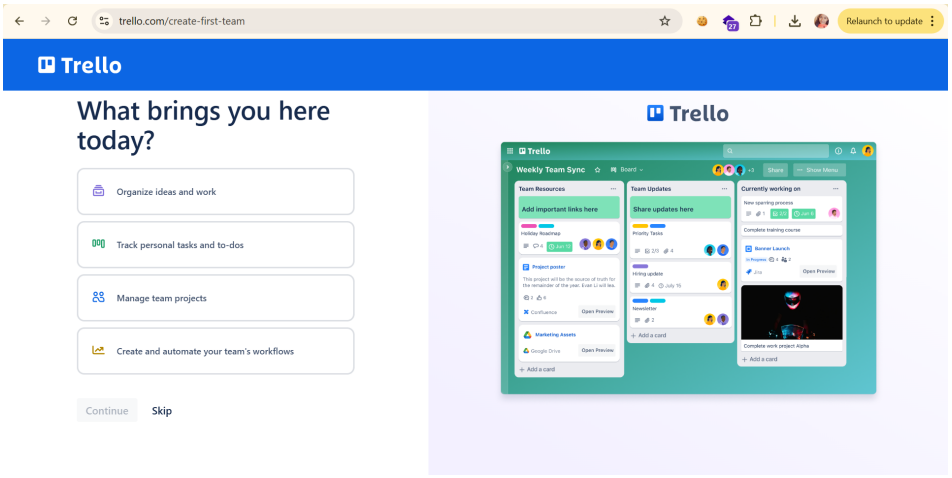
1. **Log In:**
   - Go to the Trello website.
   - Enter your credentials to log in.
2. **Create a Workspace:**
   - Click on the "Create new board" button on your dashboard.
   - Select "Create Workspace."
   - Fill in the workspace name and adjust settings as needed.
   - Click "Create Workspace."
3. **Create a Board:**
   - Within your new workspace, click on "Create Board."
   - Name your board and select a background color or image if desired.
   - Click "Create Board."
4. **Delete a Board:**
   - Open the board you want to delete.
   - Click on the "Show Menu" button on the right side.
   - Select "More," then click "Close Board."
   - After closing, you'll have the option to delete the board permanently.



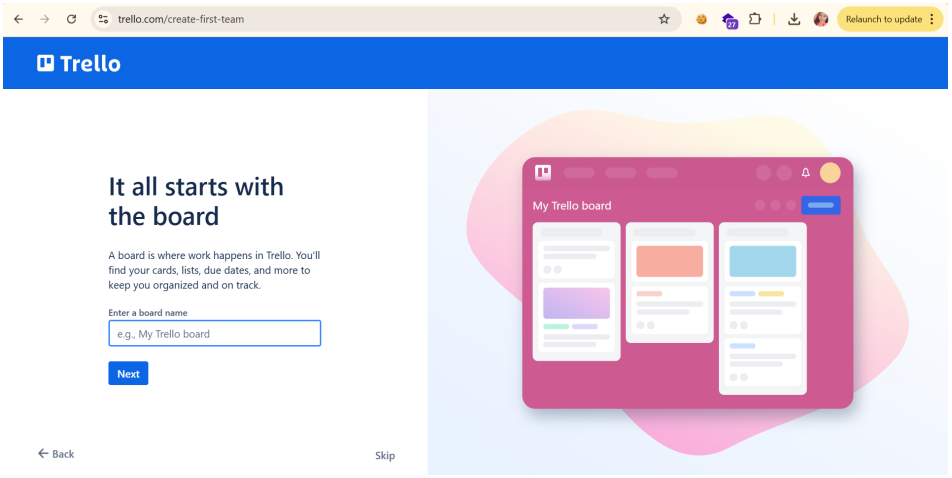
click on sign up its free button



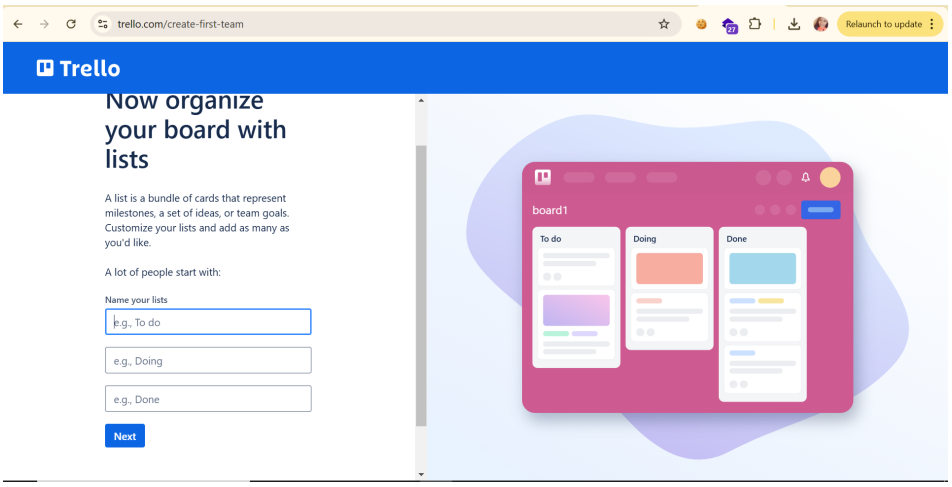
sign in with google



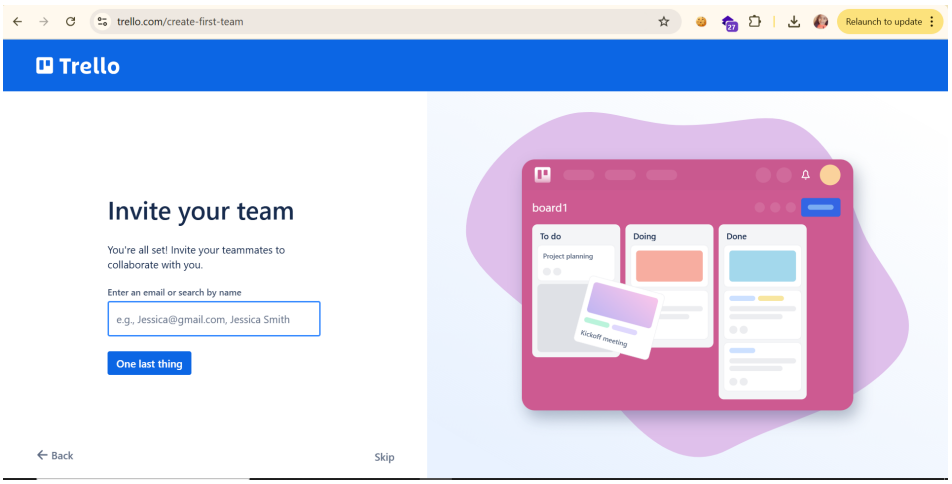
click on skip

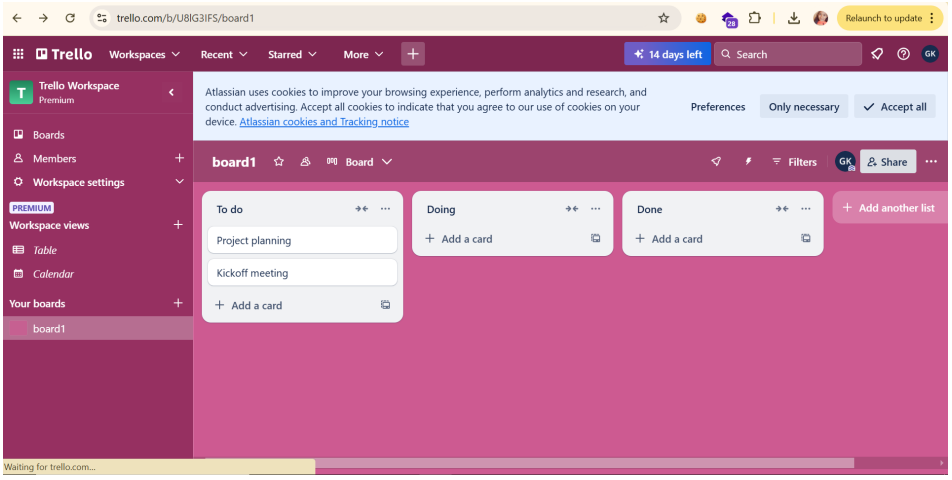


enter a board name

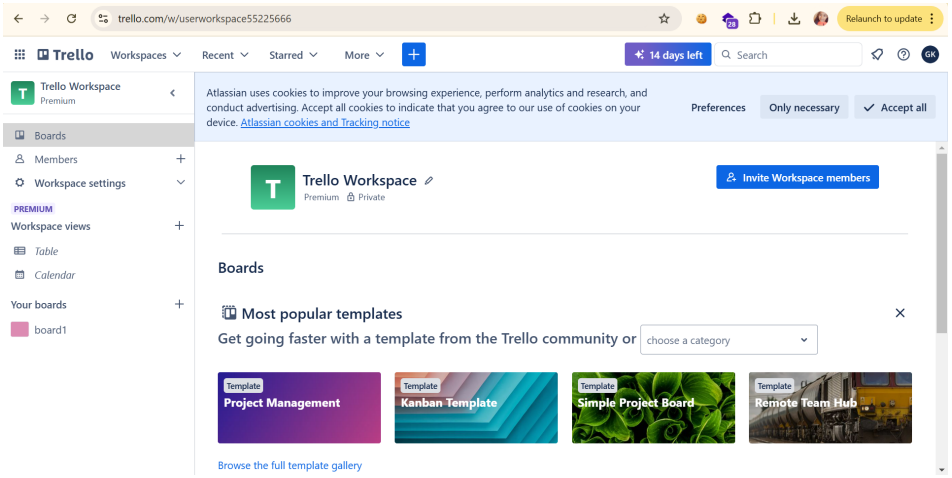


click on skip

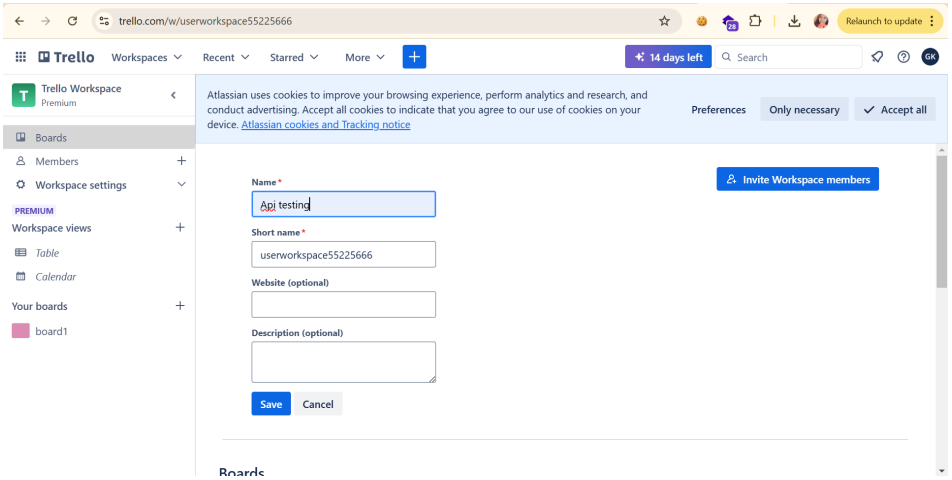




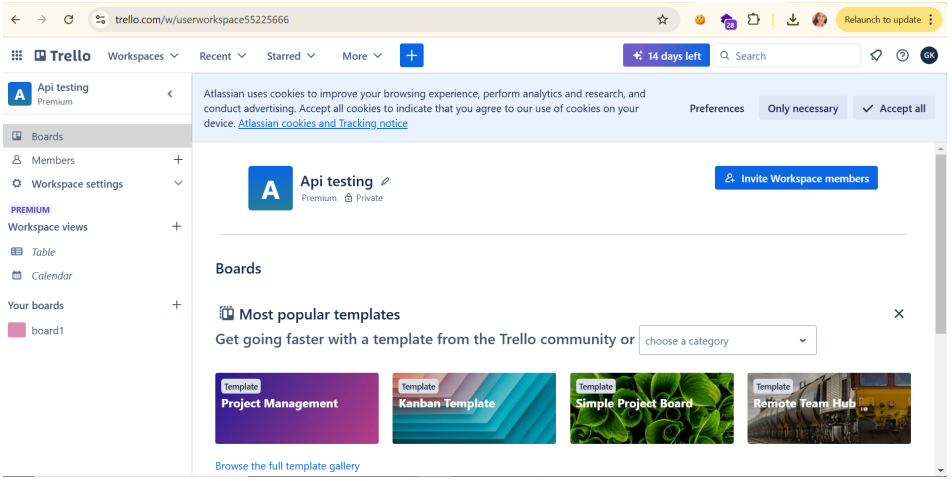
click on workspaces and select the trello workplace



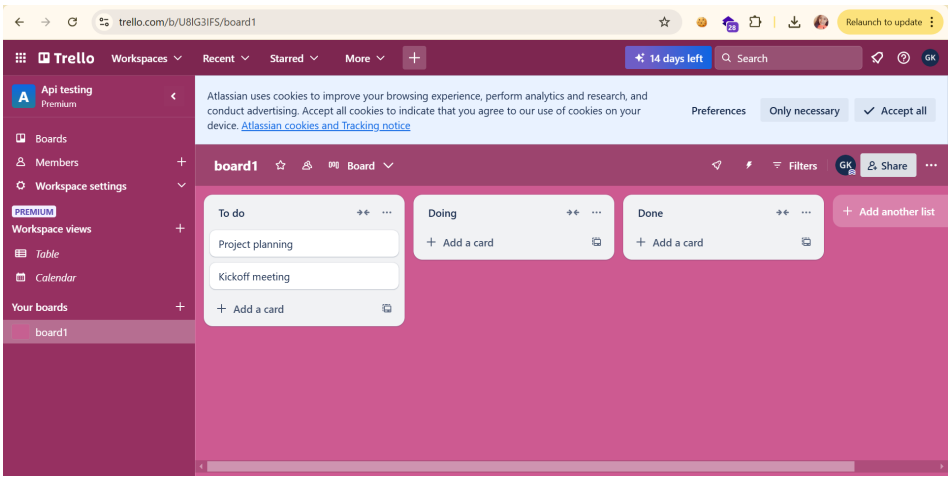
if you want to edit the workplace name you can edit



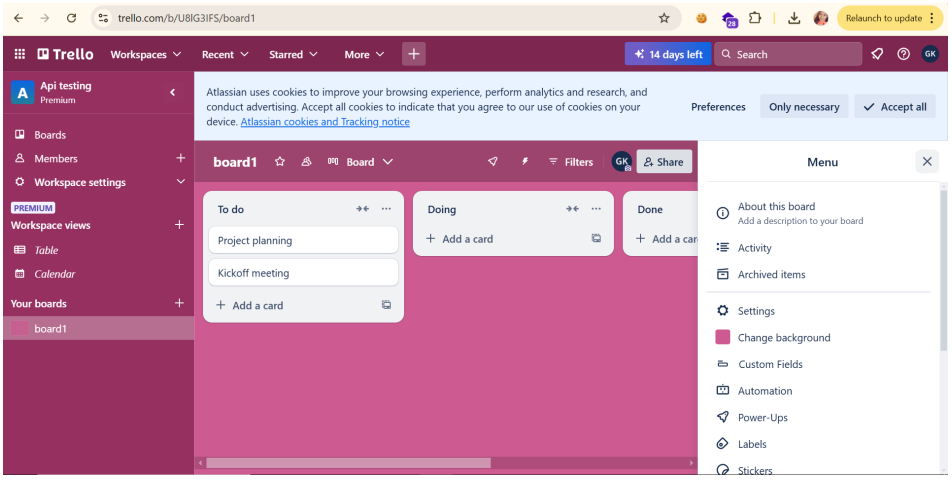
click on save



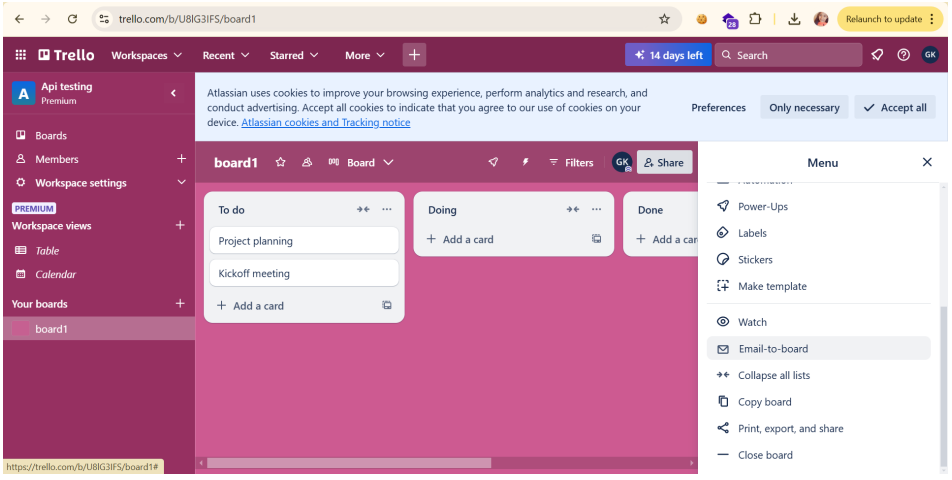
go to boards :

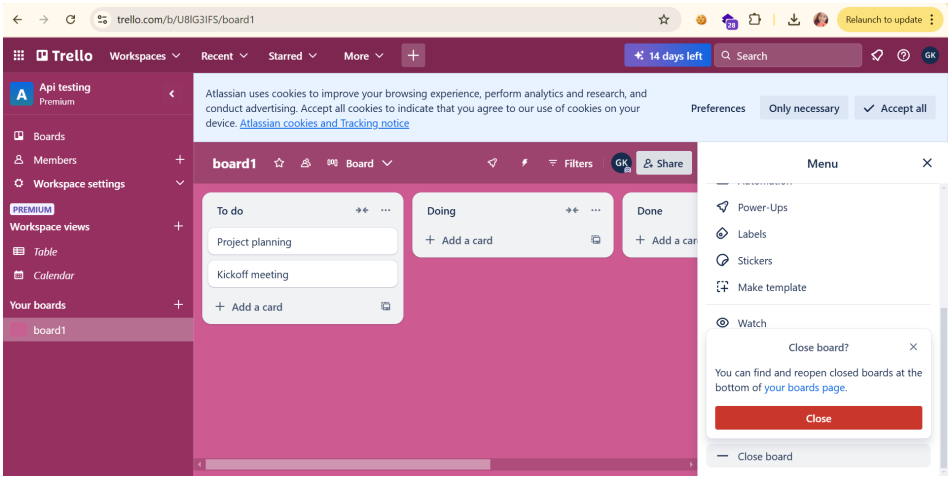


click on three dots

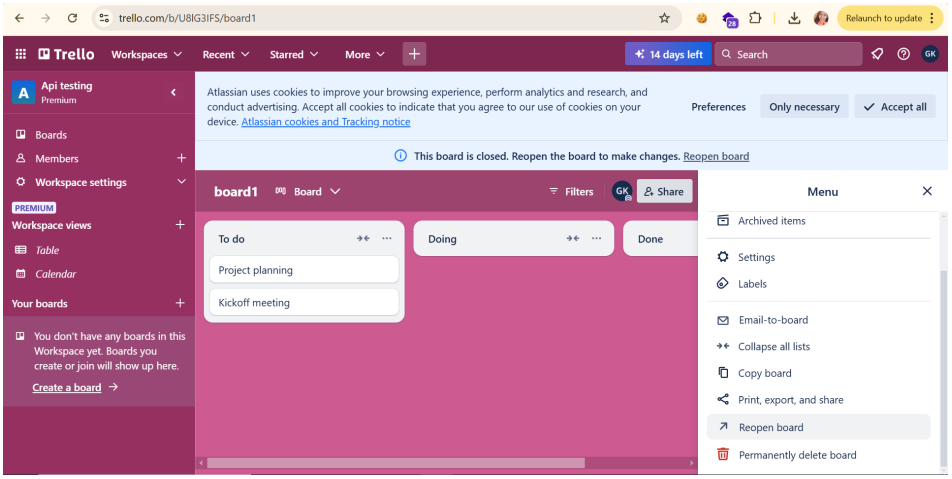


click on close board

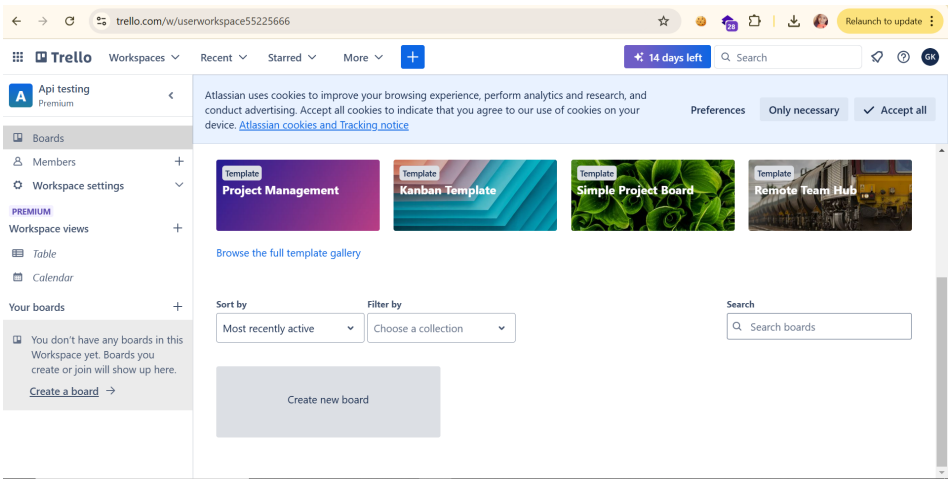




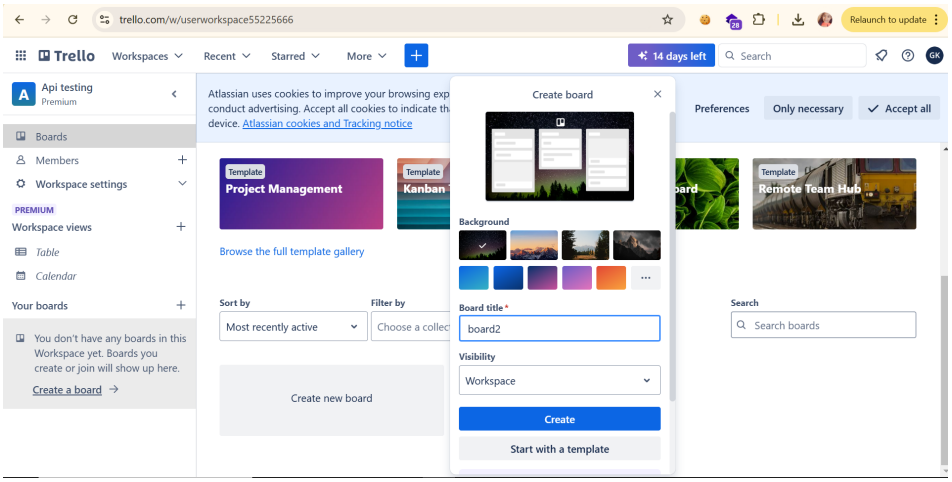
after clicking close click on permanently delete board



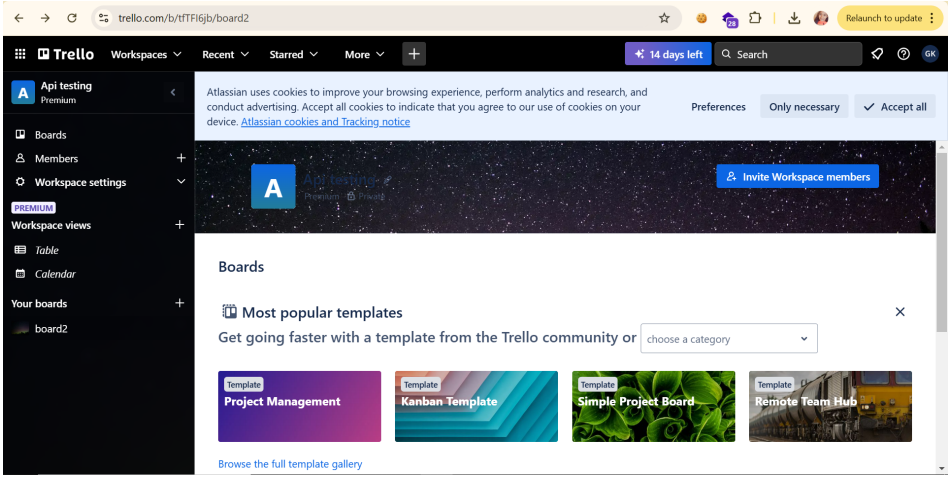
if you want to create new board click on create new board

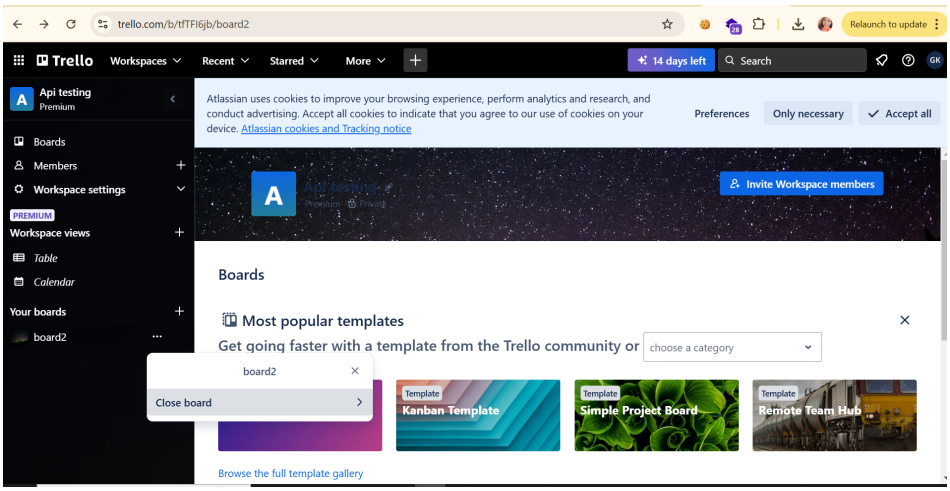


give the board title and create

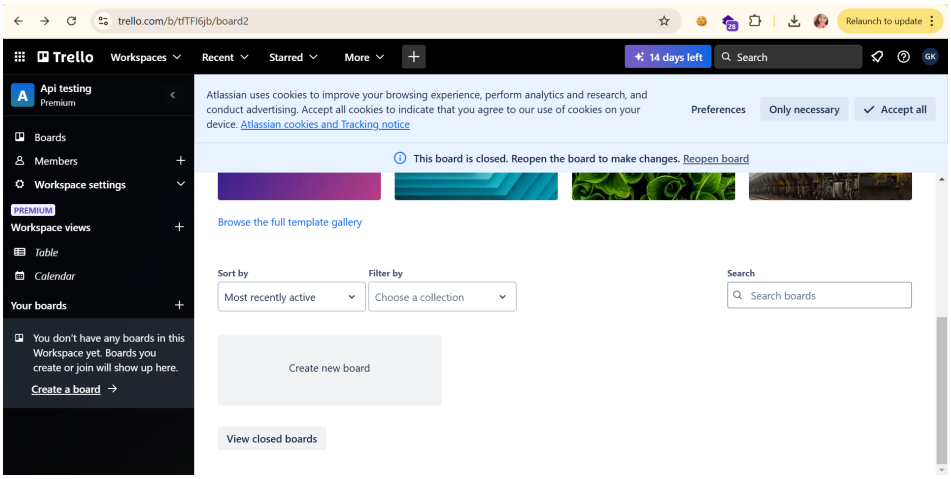


go to your boards and select board2 and click on three dots and close the board

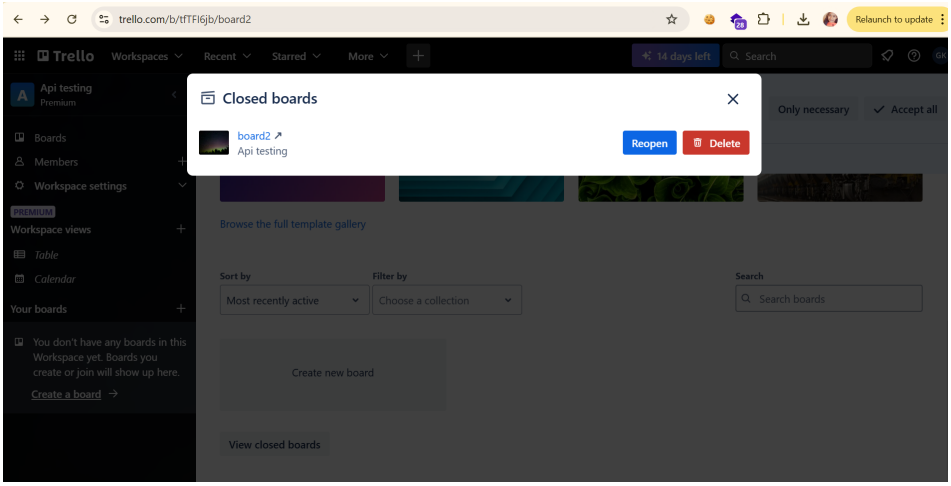




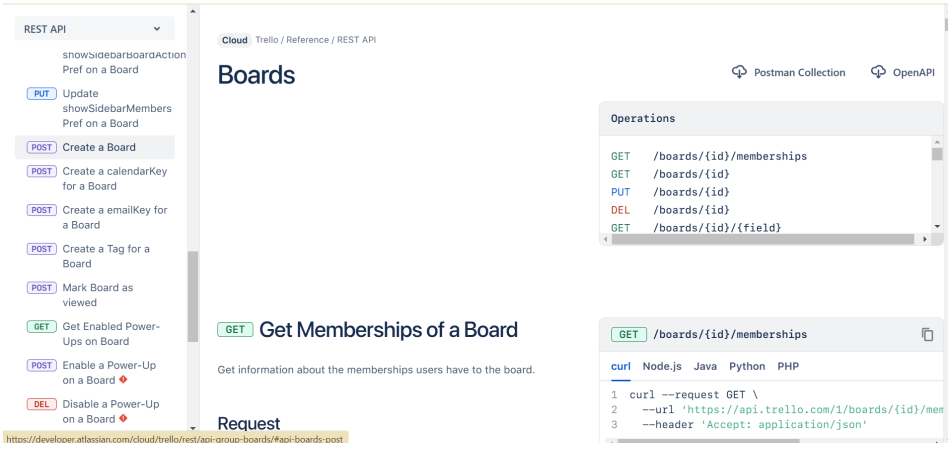
if you want to delete board permanently then click on view closed boards

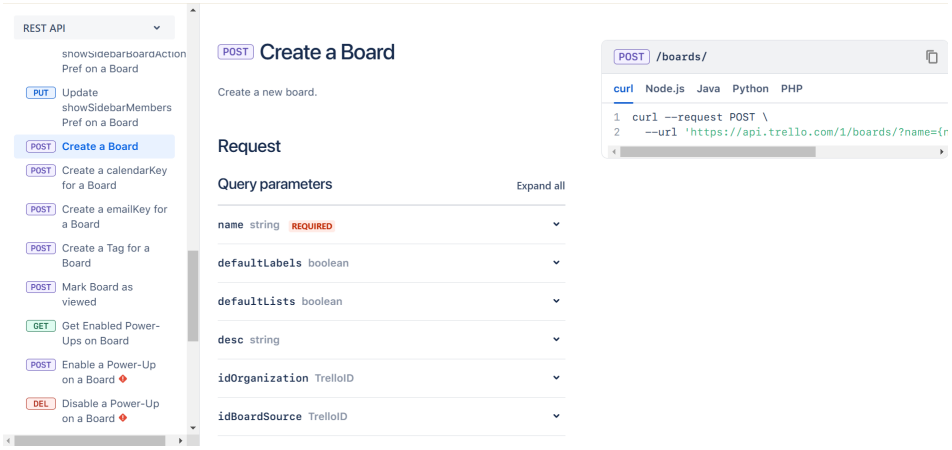


click on delete



1. Go to this url : https://developer.atlassian.com/cloud/trello/rest/api-group-actions/#api-group-actions
2. click on boards and click on create a board





https://api.trello.com/1/boards/?name={name}&key=APIKey&token=APIToken

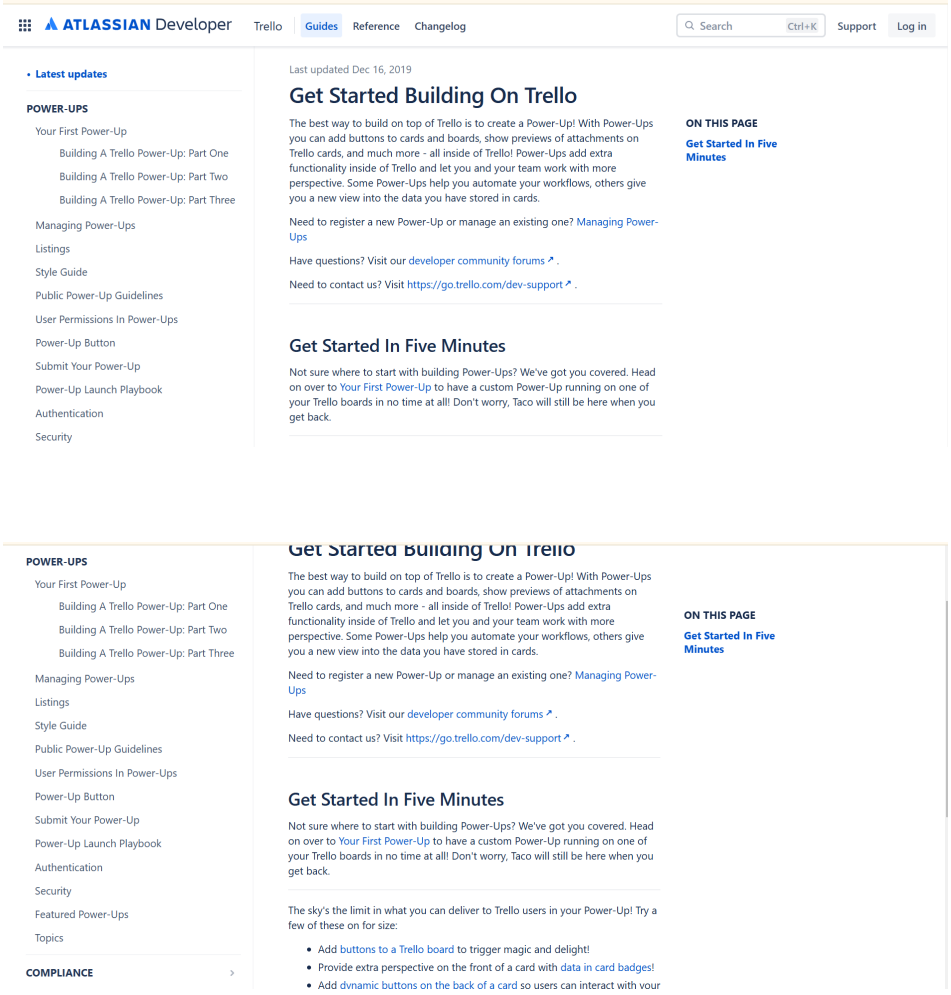
the above url is the one we are going to use

in name ={name} ==> give any name

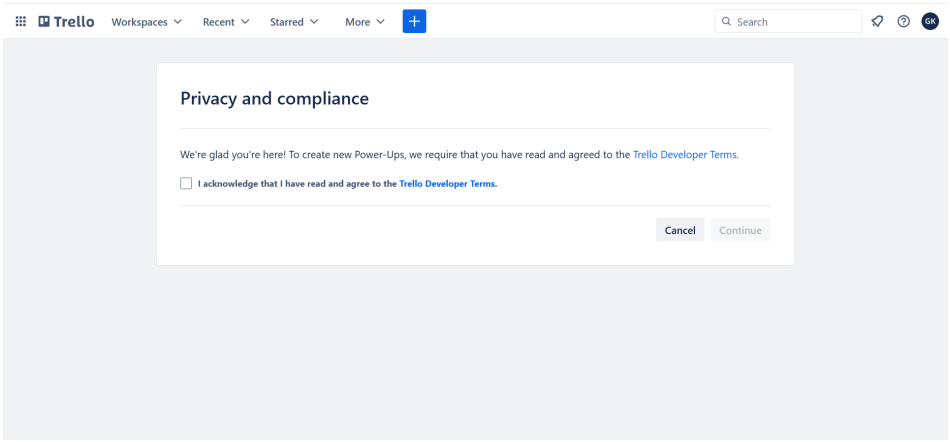
name = geethika

now how to get api key and api token

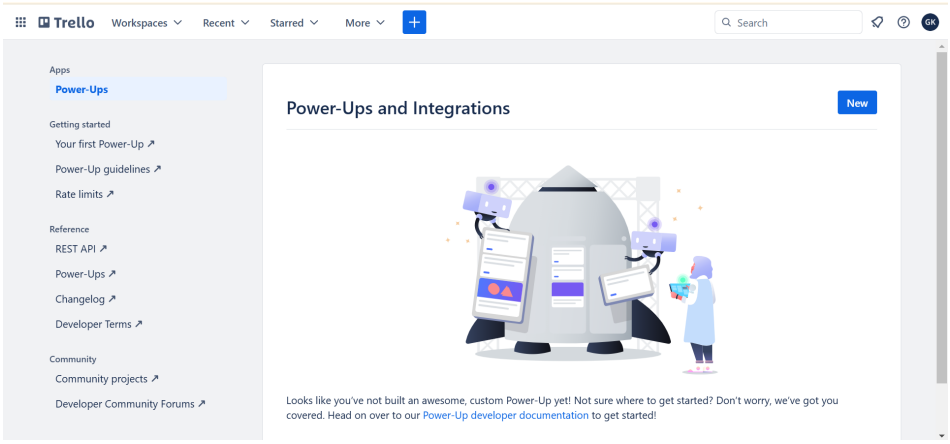
1. click on guides from this document
2. go to power ups and click on security



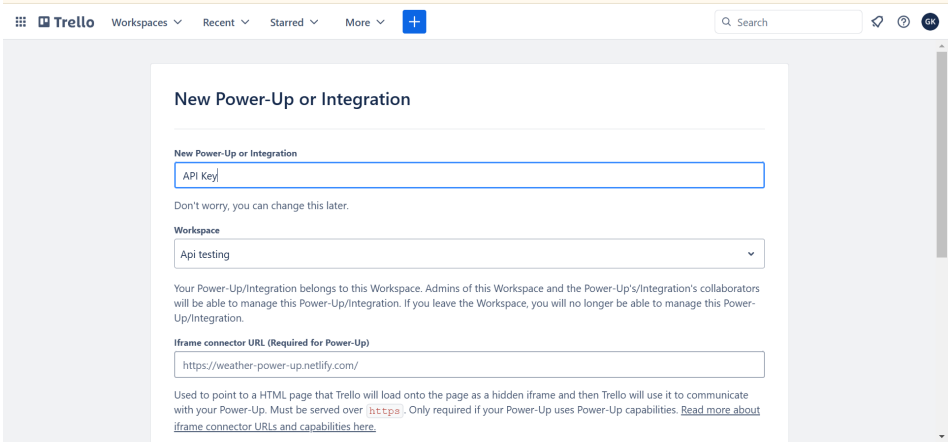
click on acknowledge and click on continue



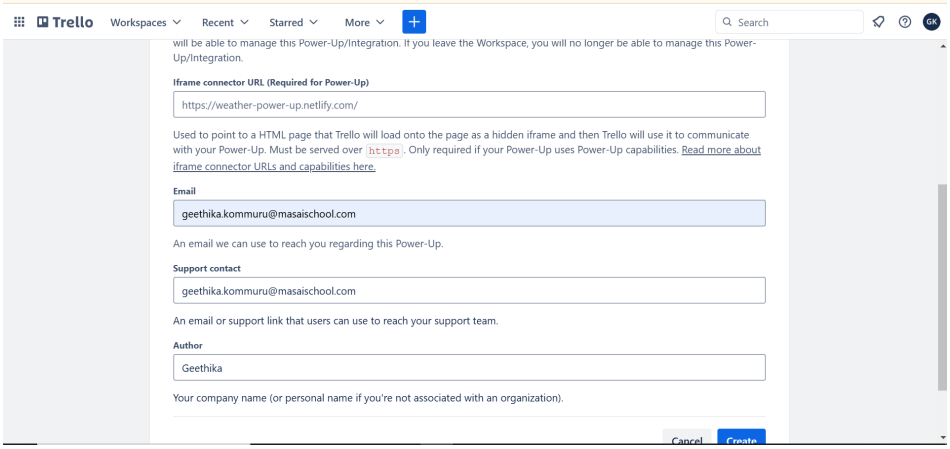
click on new



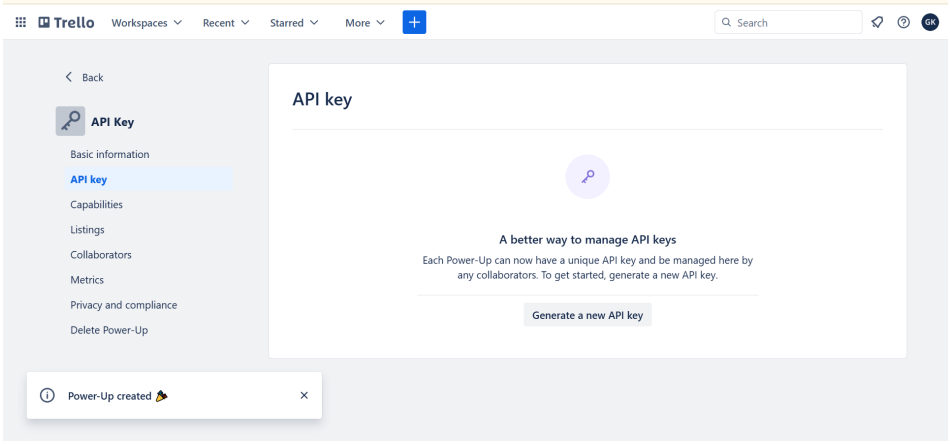
give the name and select the workspace



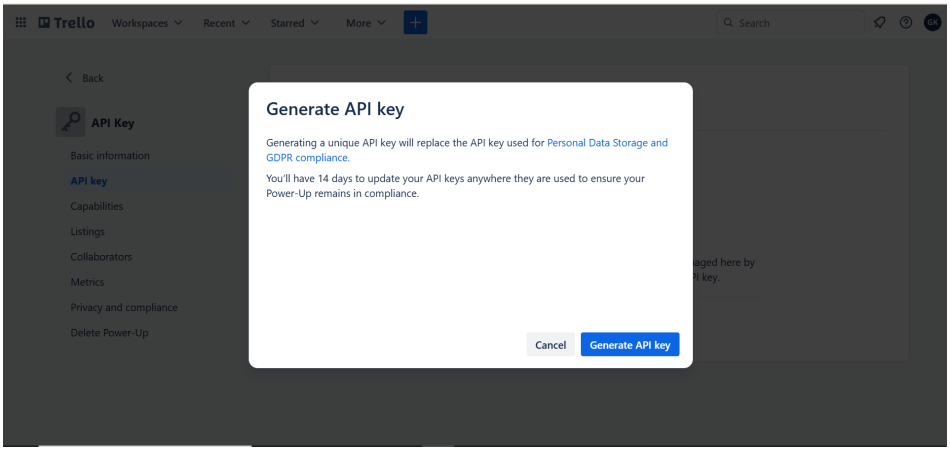
provide the email and support contact, author and click on create

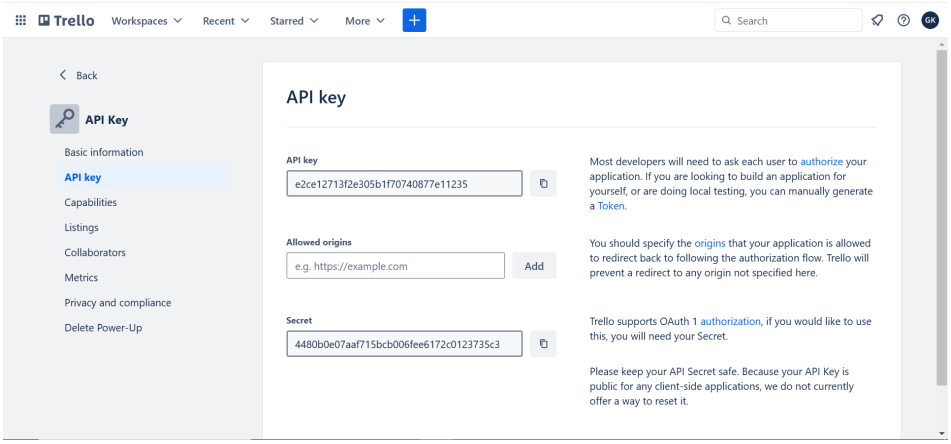


click on generate a new api key



click on Generate API key

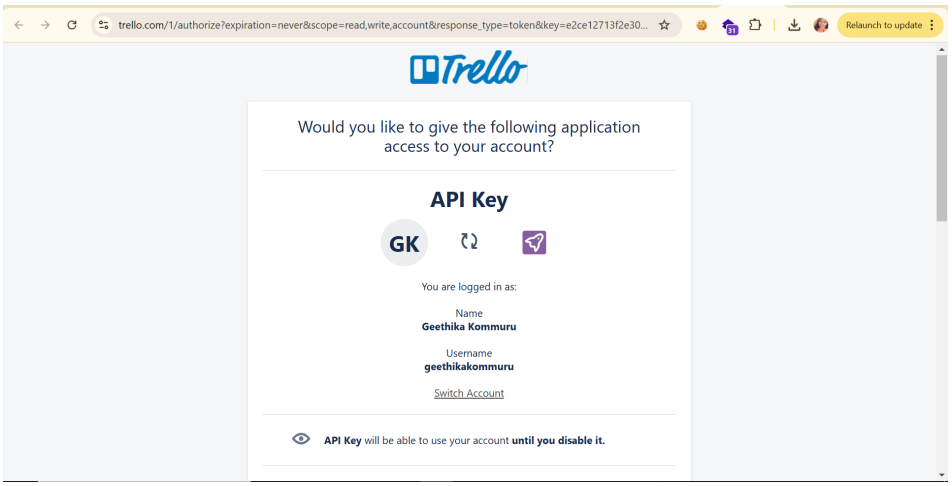




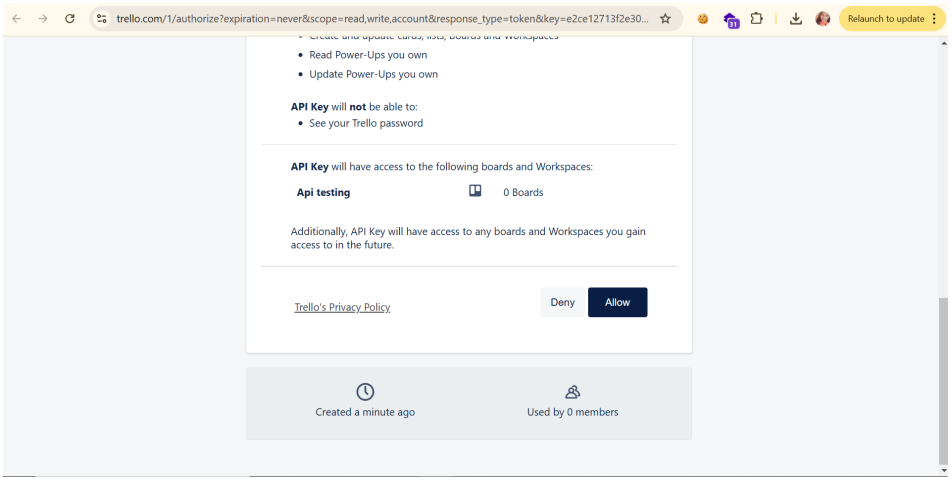
copy the api key

e2ce12713f2e305b1f70740877e11235

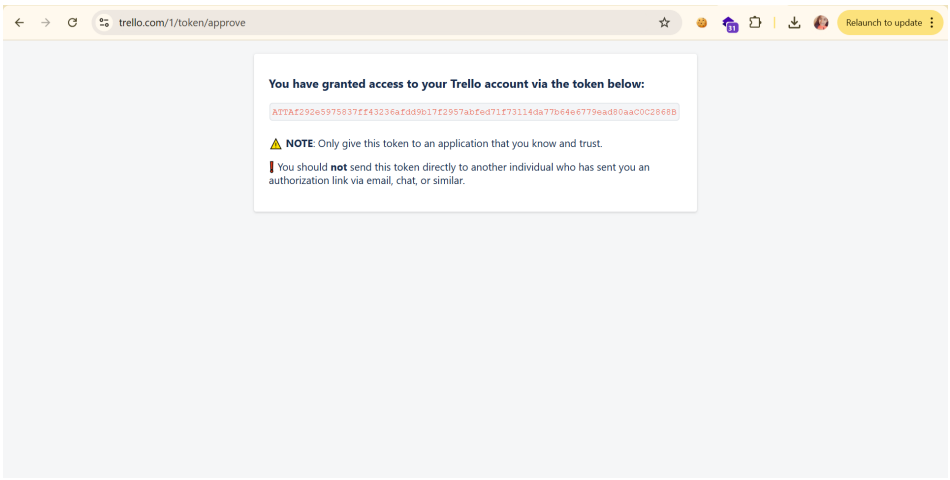
for token click on token on right side.



click on allow



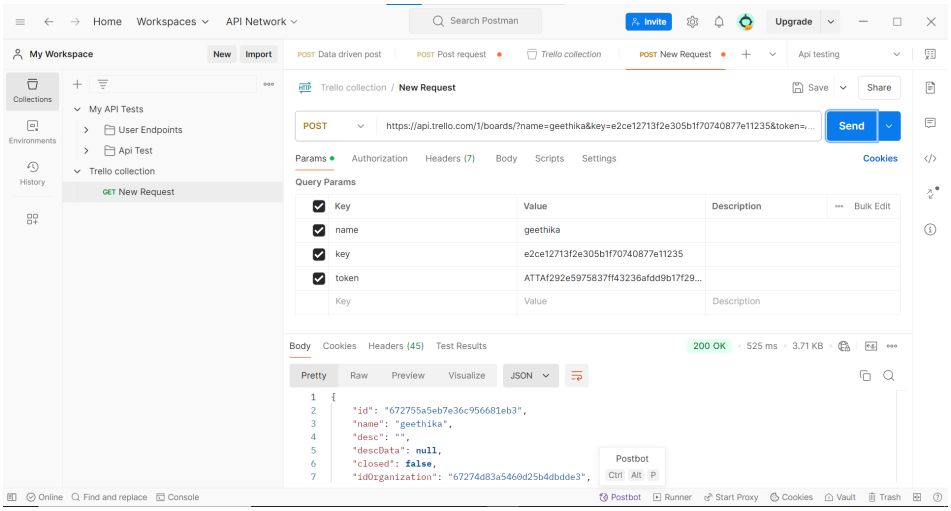
copy the token



Token - ATTAf292e5975837ff43236afdd9b17f2957abfed71f73114da77b64e6779ead80aaC0C2868B

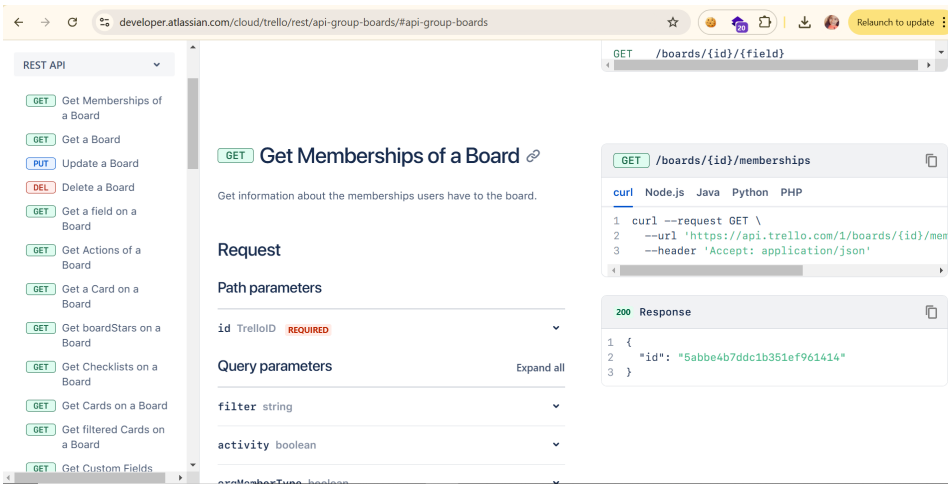
Create a board:

1. URL : https://api.trello.com/1/boards/?name={name}&key=APIKey&token=APIToken
2. Give name, api key and access token
3. **https://api.trello.com/1/boards/?name=geethika&key=e2ce12713f2e305b1f70740877e11235&token=ATTAf292e5975837ff43236afdd9b17f2957abfe**
4. now go to postman and create a collection and after that create a post request with this url :



GET A BOARD:

Click on get a board

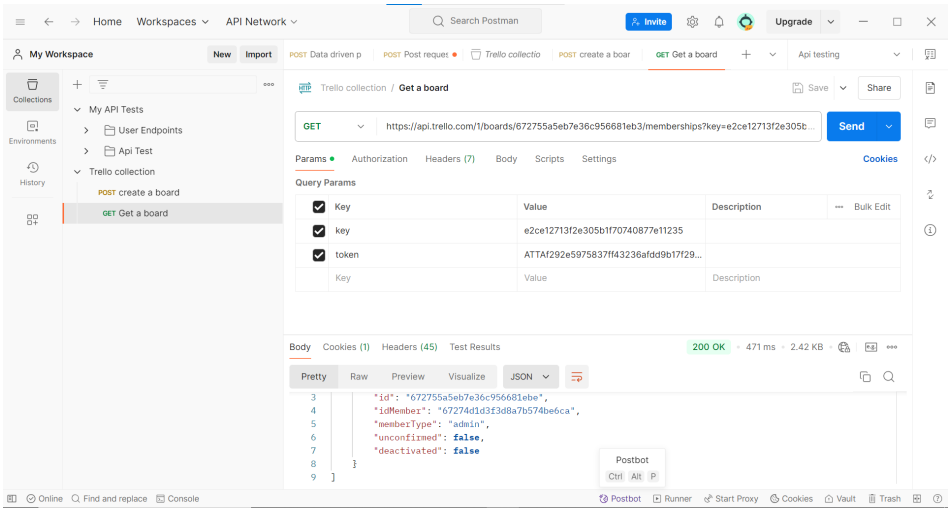


copy the url :

https://api.trello.com/1/boards/{id}/memberships?key=APIKey&token=APIToken

paste the id from create a board request and api key and token

https://api.trello.com/1/boards/672755a5eb7e36c956681eb3/memberships?key=**e2ce12713f2e305b1f70740877e11235**&token=ATTAf292e5975837ff43236afdd9b17f2957abfed71f73114da77b64e6779



in REST ASSURED :

```
package Api_Testing;

import static io.restassured.RestAssured.given;

import org.json.simple.JSONObject;
import org.testng.annotations.Test;

public class Trello {
```

```
    @Test
    public void createboard() {
        given()
            .queryParam("name", "Board123")
            .queryParam("key", "e2ce12713f2e305b1f70740877e11235")
            .queryParam("token", "ATTAf292e5975837ff43236afdd9b17f2957abfed71f73114da77b64e6779e
            .contentType("application/json")
        .when()
            .post("https://api.trello.com/1/boards/")
        .then()
            .statusCode(200)
            .log().all();
    }
}
```

```
package Api_Testing;

import static io.restassured.RestAssured.given;

import org.json.simple.JSONObject;
import org.testng.annotations.Test;

public class Trello {

    String keyString = "e2ce12713f2e305b1f70740877e11235";
    String tokenString = "ATTAf292e5975837ff43236afdd9b17f2957abfed71f73114da77b64e6779ead80aaC0
    String baseurl = "https://api.trello.com";
    @Test
    public void createboard() {
        given()
            .queryParam("name", "Board123")
            .queryParam("key", "e2ce12713f2e305b1f70740877e11235")
            .queryParam("token", "ATTAf292e5975837ff43236afdd9b17f2957abfed71f73114da77b64e6779e
            .contentType("application/json")
        .when()
            .post("https://api.trello.com/1/boards/")
        .then()
            .statusCode(200)
            .log().all();
    }
    @Test
    public void Get_A_Board() {
        String idString = "672782d1a806228375f30d64";
        given()
        .get(baseurl+"/1/boards/"+idString+"/?key="+keyString+"&token="+tokenString)
        .then()
        .log().all();
    }
}
```

```java
package Api_Testing;

import static io.restassured.RestAssured.given;

import org.json.simple.JSONObject;
import org.testng.annotations.Test;

public class Trello {

    String keyString = "e2ce12713f2e305b1f70740877e11235";
    String tokenString = "ATTAf292e5975837ff43236afdd9b17f2957abfed71f73114da77b64e6779ead80aaC0
    String baseurl = "https://api.trello.com";
    @Test
    public void createboard() {
        given()
            .queryParam("name", "Board123")
            .queryParam("key", "e2ce12713f2e305b1f70740877e11235")
            .queryParam("token", "ATTAf292e5975837ff43236afdd9b17f2957abfed71f73114da77b64e6779e
            .contentType("application/json")
        .when()
            .post("https://api.trello.com/1/boards/")
        .then()
            .statusCode(200)
            .log().all();
    }
    @Test
    public void Get_A_Board() {
        String idString = "672782d1a806228375f30d64";
        given()
        .get(baseurl+"/1/boards/"+idString+"/?key="+keyString+"&token="+tokenString)
        .then()
        .log().all();
    }
    @Test
    public void delete_A_Board() {
        String idString = "672779c186ae12ff5bce1505";
        given()
        .delete(baseurl+"/1/boards/"+idString+"?key="+keyString+"&token="+tokenString)
        .then()
        .log().all();
    }
}
```

8. Summary

In this guide, we covered REST Assured, a powerful Java library for testing REST APIs. We discussed its setup and provided examples to validate API responses for both GET and POST requests. With REST Assured's intuitive syntax and powerful assertion methods, developers can quickly build reliable API tests that verify both the structure and content of HTTP responses. Using REST Assured effectively can help ensure API reliability and improve the robustness of application functionality.